Programming Techniques
G. Manacher, S.L. Graham Editors

An Efficient List-Moving Algorithm Using Constant Workspace

Douglas W. Clark
Carnegie-Mellon University

An efficient algorithm is presented for moving arbitrary list structures, using no storage (apart from program variables) other than that required to hold the original list and the copy. The original list is destroyed as it is moved. No mark bits are necessary, but pointers to the copy must be distinguishable from pointers to the original. The algorithm is superior in execution speed to previous algorithms for the same problem. Some variations and extensions of the algorithm are discussed.

Key Words and Phrases: list moving, list copying, LISP, space complexity, constant workspace
CR Categories: 4.34, 4.49, 5.25

This work was done while the author was supported by a grant from the Xerox Corporation Palo Alto Research Center. Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

1. Introduction

The problem addressed here is how to move an arbitrary LISP-type list structure using an amount of storage (other than that required to hold the original list and the copy) that does not grow with the size or complexity of the list. Unlike the problem of *copying* a list [5], *moving* a list allows the old list to be destroyed during processing. Following LISP conventions [7], assume that each *list cell* contains two pointers, called *car* and *cdr*, which can point to any list cell or to non-list items, which are called *atoms*.

If a stack of sufficient depth is available, moving a list structure is relatively straightforward. Fenichel and Yochelson [4] give a recursive list-moving algorithm as part of their garbage collector. Minsky's algorithm [8], also intended for use in a garbage collector, optimizes use of its stack but still needs more than a constant amount of working storage. Both of these algorithms require time proportional to the number of cells moved.

Cheney's algorithm [1] was the first that needed workspace of only constant size. Although it was originally intended for "compact lists"—i.e. those *cdr*s that point to the following location in memory are simply omitted—it can easily be adapted to work on LISP-type structures. In this paper "Cheney's algorithm" will mean the LISP-oriented version of the original. Reingold's algorithm [9] employs the Deutsch-Schorr-Waite list-tracing technique [6, p. 417; 10] to avoid using a stack. This idea was suggested by Fenichel and Yochelson [4]. Reingold's algorithm traces lists in the *car* direction; that is, it follows *car* before *cdr* if there is a choice. In the interest of uniformity, "Reingold's algorithm" will hereafter mean the simple modification of the original that traces in the *cdr* direction instead.

Cheney's and Reingold's algorithms are both linear in the number of cells moved, and both require at least two visits to each cell. The algorithm presented here is also linear, but must revisit only those cells both of whose pointers point to lists. A recent empirical study of list structure data in LISP [2] found that in the programs considered, about one third of *car*s and three fourths of *cdr*s were lists. Assuming that *car*s and *cdr*s are independent as to their data types, this means that in real list structures only about one-fourth of list cells would need to be visited twice by the present algorithm.

2. An Example

The operation of the algorithm is illustrated by example in Figure 1. The list structure to be moved is $((A)B(A)C(D))$, where the two occurrences of the sublist (A) are in fact the same cell. Figure 1(a) shows the original structure. In the figure, *car* is the left-hand

pointer of a cell, and *cdr* the right-hand one. A diagonal slash designates the list-terminating atom *NIL*.

The algorithm first copies the top level of the list, with some changes, into sequential locations in the new list area. The state of affairs after this has been done is shown in Figure 1(b), in which the following conditions hold:

(1) Old *car*s have been replaced by "forwarding addresses": *car*(x) is x's new location. This technique, or some variant of it, is employed by all of the other list-moving and copying algorithms discussed here [1, 3, 4, 5, 8, 9]. Discovery of a forwarding address where an ordinary *car* was expected inhibits the creation of spurious copies of shared cells.

(2) In the copy, atomic *car*s and the one atomic *cdr* (so far) have their final values.

(3) List *car*s in the copy point to the original sublists in the old list area.

(4) List *cdr*s in the copy have their final values, and all point to the next consecutive cell in memory.

(5) Old top-level cells with list *car*s have been linked together through their *cdr*s in LIFO order on a list k. The first such cell encountered terminates k by having *NIL* in its *cdr*.

The algorithm must now, in effect, "pop" the k-list and move the old sublist pointed to by *car* of the copy of the first cell on k, namely, *car*(*car*(k)). The top level of this sublist will be moved as described above; and as more cells with list *car*s are encountered, they will be attached to the front of k. Each time k is popped, the corresponding list *car* in the copy will be set to its final value. This value will be the next free cell in the new area if the sublist has not already been moved, or the forwarding address of the sublist, *car*(*car*(*car*(k))), if it has.

Figure 1(c) shows the state of things after both sublists of the original structure have been copied. One cell remains on the k-list. The forwarding address left in the old shared sublist (A) demonstrates its usefulness by preventing creation of a second copy. Figure 1(d) shows the final result.

It is in its use of the k-list that the present algorithm differs most significantly from those of Cheney and Reingold. Cheney's algorithm, after copying the top level of a list much as is done here, visits sequentially each cell of the copy to find sublists. Reingold's algorithm attaches *all* visited cells to its version of the k-list during copying, and computes final values for new list *cdr*s only when backtracking up the list. Both algorithms thus revisit cells (or copies of cells) with atomic *car*s.

3. The Algorithm

Although the algorithm does not require a mark bit in each cell, it does need to be able to tell whether a pointer points into the new list area. Assuming the new region to be a block of contiguous locations, this could be done simply by comparing the pointer with the address boundaries of the region. Let the predicate *new*(x) be true if and only if x points within the new area. Let the free variable n point to the first available cell in the new area, and assume that each list cell occupies one word of the sequentially addressed memory. The algorithm given below will move the list pointed to by h from the old list region to the new. On termination of the algorithm, h will point to the new list, and n to the next free cell in the new area.

Part A. Copy the top level of a list.

A1. [Initialize.] $x \leftarrow h$, $h \leftarrow n$, and $k \leftarrow NIL$.

A2. [Save *car* and *cdr*.] $a \leftarrow car(x)$ and $d \leftarrow cdr(x)$.

A3. [Store forwarding address.] $car(x) \leftarrow n$.

A4. [Is *car* a list?] If a is a list then $cdr(x) \leftarrow k$ and $k \leftarrow x$. (k points to the most recently visited cell whose *car* is a list.)

A5. [Copy old *car*.] $car(n) \leftarrow a$. (If a is an atom, $car(n)$ has its final value now.)

A6. [Compute and write new *cdr*.] If d is an atom then $cdr(n) \leftarrow d$, $n \leftarrow n+1$, and go to step B1. If $new(car(d))$ then $cdr(n) \leftarrow car(d)$, $n \leftarrow n+1$, and go to step B1. Otherwise, d must be an unvisited list, so $cdr(n) \leftarrow n+1$, $n \leftarrow n+1$, $x \leftarrow d$, and return to step A2. (In all cases $cdr(n)$ gets its final value in this step. If d is an atom or an already visited list, *cdr*-direction tracing stops and we go to Part B to find the most recently seen sublist. Otherwise, we continue *cdr*-following and return to step A2.)

Part B. Find the most recently visited sublist.

B1. [$k = NIL$?] If $k = NIL$ then the algorithm terminates with h pointing to the new list and n to the next free cell.

B2. [Remove first element of k-list.] $x \leftarrow car(car(k))$, $t \leftarrow k$, and $k \leftarrow cdr(k)$.

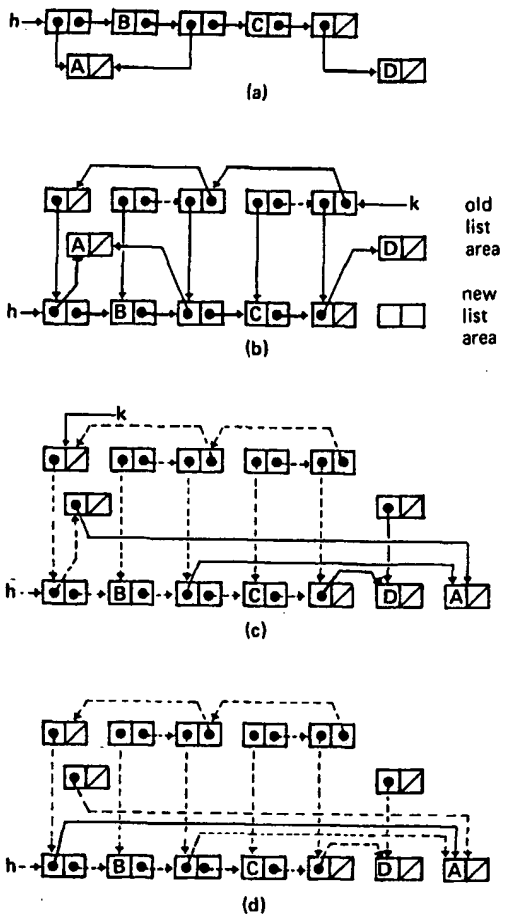
B3. [Compute and write new *car*.] If $new(car(x))$ then $car(car(t)) \leftarrow car(x)$ and return to step B1. Otherwise, $car(car(t)) \leftarrow n$ and go to step A2. (*Car* of the copy of the cell just removed from k gets its final value in this step.)

4. Improvements, Variations, and Extensions

The algorithm as it stands puts onto the k-list cells with list *car*s and atomic *cdr*s (or already copied *cdr*s), only to remove such cells immediately in order to copy the sublist. An improvement in the algorithm, therefore, would be to avoid these undesirable second visits by adding a cell to k if and only if *car* is a list and *cdr* is an unvisited list. Since *cdr* must be checked for this property anyway (to establish its new value), this speed-up would require no additional computational effort.

If this change is made, then every cell on the k-list will have the following property: *cdr* of the copy of the cell will be the next sequential cell in the new list area. This redundancy makes it possible to keep the k-list *in the copy* rather than in the original list, since the *cdr*s temporarily displaced by the links of k can easily be recomputed during the second visit to each cell. Keeping k in the copy eliminates one memory fetch when each element of k is removed: *car*(k) is desired

Fig. 1. Moving the list ((*A*) *B* (*A*) *C* (*D*)): (a) initial structure; (b) and (c) during processing; (d) final structure. Dotted pointers are those unchanged from the previous figure.



instead of *car*(*car*(*k*)). The resulting algorithm is faster than the one given in the previous section.

If reference counts are available for each list cell, a further improvement is possible, namely, avoiding the storage of a forwarding address for cells with a reference count of one. (This clearly requires that the *k*-list be kept in the copy.) This idea was used by Deutsch and Bobrow in their linearization algorithm [3]. If, moreover, *all* cells have a reference count of one, then the list can be moved without altering the original structure.

If many list cells are pointed to more than once, it may be profitable to evaluate *new*(*car*(*car*(*x*))) when a cell *x* with a list *car* is *first* visited. If *car*(*car*(*x*)) is a new list pointer (as it would be, in this case, much of the time), then *x* need not be added to the *k*-list, thus saving the second visit and the attendant overhead. If, on the other hand, few cells are shared (as appears to be the case in real LISP programs [2]), this check would not be worthwhile. The reason is that in the usual case *new*(*car*(*car*(*x*))) would be false, *x* would be attached to the *k*-list, and when, sometime later, *x* was removed from the *k*-list, *new*(*car*(*car*(*x*))) would have to be evaluated *again* to make sure *x* had not been encountered in the interim.

The new area into which a list is moved need not be a block of contiguous locations. New cells could be acquired from an arbitrary free-list by an obvious change in the algorithm. If old and new list areas overlap in memory, however, the function *new*(*x*) would probably require a mark bit in each cell. (Clearly this change would not permit the *k*-list links to be kept in the new *cdr*s, as suggested above.)

Acknowledgments. Helpful comments on a draft of this paper were made by D.G. Bobrow, E.S. Cohen, L.P. Deutsch, and S.H. Fuller.

Received July 1975

References

1. Cheney, C.J. A nonrecursive list compacting algorithm. *Comm. ACM 13*, 11 (Nov. 1970), 677–678.
2. Clark, D.W., and Green, C.C. An empirical study of list structure in LISP. *Comm. ACM*, to appear.
3. Deutsch, L.P, and Bobrow, D.G. An efficient, incremental, automatic garbage collector. *Comm. ACM*, to appear.
4. Fenichel, R.R., and Yochelson, J.C. A LISP garbage-collector for virtual-memory computer systems. *Comm. ACM 12*, 11 (Nov. 1969), 611–612.
5. Fisher, D.A. Copying cyclic list structures in linear time using bounded workspace. *Comm. ACM 18*, 5 (May 1975), 251–252.
6. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.
7. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine—I. *Comm. ACM 3*, 4 (April 1960), 184–195.
8. Minsky, M.L. A LISP garbage collector algorithm using serial secondary storage. Artificial Intelligence Project Memo 58 (revised), M.I.T. Project MAC, Dec. 1963.
9. Reingold, E.M. A nonrecursive list moving algorithm. *Comm. ACM 16*, 5 (May 1973), 305–307.
10. Schorr, H., and Waite, W. An efficient machine-independent procedure for garbage collection in various list structures. *Comm. ACM 10*, 8 (Aug. 1967), 501–506.

Corrigendum

1975 ACM Student Award Paper: First Place

Guy L. Steele Jr., "Multiprocessing Compactifying Garbage Collection," *Comm. ACM 18*, 9 (Sept. 1975), 495–508.

P. 501: In the routine *relocate*, after the comment, "Relocate an object," the next three lines should be permuted to read:

```
munch(address(s, k));
s.cells[j]←s.cells[k];
s.cells[j].mark←true;
```

This is needed to prevent a timing error which can occur if the list processor modifies a component of the object being relocated from (using the *clobber* routine).